(e.g., a machine-readable storage medium) and perform any one or more of the methodologies discussed herein.

It will be noted that throughout the appended drawings, like features are identified by like reference numerals.

DETAILED DESCRIPTION

The description that follows describes systems, methods, techniques, instruction sequences, and computing machine program products that constitute illustrative embodiments of the disclosure. In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide an understanding of various embodiments of the inventive subject matter. It will be evident, however, to those skilled in the art, that embodiments of the inventive subject matter may be practiced without these specific details.

Current object-oriented programing is not optimized for performance due in part to the use of reference value objects that contain pointers to data rather than containing data directly. Existing game development technology often uses reference value structures to define objects within a game. This is based on the concept of an object within the object-oriented programming framework and is used for simplicity of programming (e.g., since the behavior and attributes of a programming object align well with those of a game object). However, object oriented programming may be optimized on a conceptual level and for ease of programming, but it is not always optimized for performance with respect to video game play. The main reason for the lack of optimized performance is that OOP programing does not automatically provide the optimum use of memory. OOP objects often contain pointers to data while the data itself is scattered randomly over distant memory locations. The result is that game object data is often in random places within memory and often contains pointers (e.g., to data) in other random locations within memory. In order to access the data for one or more characters (e.g., to determine the character location in a scene), a game engine will often have to access several separate random memory locations. There is also no hard guarantee of the relative location of data within memory for two different game objects. Accessing random memory locations for all game objects in a video game scene which runs at 60 frames per second (fps) or more is inefficient, especially considering the large amount of game objects which are typically in play during any given video game frame. Having game object data scattered over memory creates an inefficiency due to memory access time (e.g., the time it takes a central processing unit (CPU) to access a memory location, which is typically hundreds of CPU cycles each time a memory location is accessed). All memory accessing takes time; however, having to access memory in random distant locations requires additional time because the advantages of hardware prefetching are negated. The additional time it takes to access the scattered data within memory lowers the performance of executed game code at runtime. This puts limitations, for a given CPU speed, on the number of game objects that can be active in a frame during game play if a frame rate is to be maintained (e.g., 60 frames per second for typical games). This is particularly important for virtual reality applications which require 90 frames per second for minimum quality visual output. Modern game design improves performance by incorporating graphical processing units (GPUs) to offload processing from the CPU, as well as multithreaded coding techniques to parallelize the processing of game data over multiple CPU/GPU cores. However, these techniques do not overcome the fundamental issue of accessing separate random memory locations for game objects.

Game performance can also be improved by considering data oriented programming methodology as opposed to object oriented programming methodology, however, data oriented programming requires a high degree of knowledge for a game developer, and is done manually, and is specifically targeted to each game. This is out of reach for a large portion of game developers and game designers who have only a basic knowledge of programming methodology.

Methods and apparatuses to improve the performance of a video game engine using an Entity Component System (ECS) are described herein. In accordance with an embodiment, the ECS eliminates (e.g., during game development and at runtime) the use of OOP reference value structures (e.g., pointers) to define game objects. Instead, the ECS defines game objects with data value structures (e.g., a 'struct' from C#) which do not use pointers to store data. In this sense, a same object as described herein is not an 'object' as defined within object oriented programming framework; accordingly, a game object as described herein (e.g., within the ECS) is referred to as an 'entity'.

In accordance with an embodiment, the ECS creates and uses entitles which are constructed entirely using value data types (e.g., structs in C# which do not use pointers). An entity is a collection of data that is used to represent anything in a video game, including characters, guns, treasures, trees, backgrounds, animation, effects (e.g., video and sound), 3D points, and more. The ECS groups a plurality of entities into an archetype wherein the entities share similar attributes (e.g., components as described herein) and memory layout. The ECS constructs the entities (e.g., including the components therein) within a memory in a densely packed and linear way. The ECS constantly monitors (e.g., during game play) entities within a game and adjusts the entity distribution (e.g., including the data therein) within the memory so that a maximum density of memory usage is maintained in real time as the game is being played thus allowing for high performance due to efficient memory access (e.g., using hardware prefetching) and multithreading. The ECS system provides high performance for game situations that include a large number (e.g., hundreds or thousands) of similar game objects (e.g., non-player characters, rockets, spaceships, etc.).

Turning now to the drawings, systems and methods for an Entity Component System (ECS) which is configured to provide high processing performance for a video game engine (e.g., to display video games or simulations) in accordance with embodiments of the invention are illustrated. In accordance with an embodiment, FIG. **1** shows an example entity component system **100** configured to provide ECS functionality. The ECS includes an ECS device **101** which includes one or more central processing units **104** (CPUs), and graphics processing units **106** (CPUs). The CPU **104** is any type of processor, processor assembly comprising multiple processing elements (not shown), having access to a memory **102** to retrieve instructions stored thereon, and execute such instructions. Upon execution of such instructions, the instructions cause the ECS device **101** to perform a series of tasks as described herein. The CPU can include a cache memory **105** within the CPU.

The ECS device **101** also includes one or more input devices **108** such as, for example, a keyboard or keypad, mouse, pointing device, and touchscreen. The ECS device **101** further includes one or more display devices **110**, such as a computer monitor, a touchscreen, and a head mounted display (HMD), which may be configured to display a video